

Ripple: Reflection Analysis for Android Apps in Incomplete Information Environments

Yifei Zhang, Tian Tan, Yue Li and Jingling Xue
Programming Languages and Compilers Group
School of Computer Science and Engineering, UNSW Australia
{yzhang, tiantan, yueli, jingling}@cse.unsw.edu.au

ABSTRACT

Despite its widespread use in Android apps, reflection poses graving problems for static security analysis. Currently, string inference is applied to handle reflection, resulting in significantly missed security vulnerabilities. In this paper, we bring forward the ubiquity of incomplete information environments (IIEs) for Android apps, where some critical data-flows are missing during static analysis, and the need for resolving reflective calls under IIEs. We present RIPPLe, the first IIE-aware static reflection analysis for Android apps that resolves reflective calls more soundly than string inference. Validation with 17 popular Android apps from Google Play demonstrates the effectiveness of RIPPLe in discovering reflective targets with a low false positive rate. As a result, RIPPLe enables FlowDroid, a taint analysis for Android apps, to find hundreds of sensitive data leakages that would otherwise be missed. As a fundamental analysis, RIPPLe will be valuable for many security analysis clients, since more program behaviors can now be analyzed under IIEs.

CCS Concepts

•Security and privacy → Mobile platform security; Software security engineering; Malware and its mitigation; •Theory of computation → Program analysis;

Keywords

Android, Reflection Analysis, Pointer Analysis

1. INTRODUCTION

Android’s increase in popularity and its openness have triggered a great rise in malware-spreading apps. Static analysis is a fundamental tool for detecting security threats in Android apps early at software development time at significantly reduced cost. This approach is immune to emulation detection and can provide a comprehensive picture of an app’s possible behaviors. Therefore, static analysis finds diverse applications, including data leakage detection [5, 11, 13, 15], repackaging attack detection [8, 49], security policy verification [6, 10, 30], security vetting [14, 22, 43], privacy violations [36], and malware detection [3, 9, 48].

However, reflection, under which a class, method or field can be manipulated by its string name, poses graving problems for static analysis. According to a recent study [32], malware authors can use reflection to hide malicious behaviors from detection by all the 10 commercial static analyzers tested. Similarly, academic static analyzers either ignore

reflection [12, 28, 29, 46] or handle it only partially [5], resulting in also significantly missed program behaviors.

Reflection analysis aims to discover statically the reflective targets referred to at reflective calls. For Android apps, regular string inference is currently performed to discover the string constants used as class/method/field names at reflective calls [6, 10, 17]. For example, if `cName` in `clz = Class.forName(cName)` is statically discovered to be “A”, then we know that `v` points to an object of type `A` reflectively created at `v = clz.newInstance()`. If `cName` is statically unknown, then `v = clz.newInstance()` is ignored.

Regular string inference is inadequate for framework-based and event-driven Android apps. In practice, reflection analysis must be performed together with many other analyses, including pointer analysis [13, 18, 19, 21, 37], inter-component communication (ICC) analysis [27–29], callback analysis [5, 13, 43, 45], and library summary generation [4, 7, 13]. Soundness, which demands over-approximation, is often sacrificed in order to achieve efficiency and precision trade-offs. As a result, class/method/field names used at reflective calls may be non-constant (either non-null but statically unknown or simply null). Similarly, the receiver objects at reflectively method call sites (i.e., the objects pointed to by `v` in `Method.invoke(v, ...)`) may be non-null but with statically unknown types or simply null. Such information can be missing due to, for example, unsound library summaries, unmodeled Android services, code obfuscation, and unsound handling of hard-to-analyzed Android features such as ICC, callbacks and built-in containers. In this case, regular string inference, which keeps track of only constant strings, is ineffective, resulting in missed program behaviors.

In this paper, we bring forward the ubiquity of incomplete information environments (IIEs) for Android apps, where some critical data-flows are inevitably missing during static analysis. As discussed above, these include not only the case when class/method/field names are non-null but statically unknown, which is studied previously for Java programs [18, 19, 21, 37], but also the case when these string names are null, which is investigated for the first time for Android apps in this paper. We therefore emphasize the need for resolving reflective calls in Android apps under IIEs. To this end, we introduce RIPPLe, the first IIE-aware static reflection analysis for Android apps that can resolve reflective calls more soundly than string inference at a low false positive rate. We also demonstrate its effectiveness in improving the precision of an important security analysis.

In summary, this work makes the following contributions:

- We present (for the first time) an empirical study for

IIEs in real-world Android apps, and examine some common sources of incomplete information, discuss their impact on reflection analysis, and motivate the need for developing an IIE-aware reflection analysis.

- We introduce RIPPLE, the first IIE-aware reflection analysis for Android apps, which performs type inference automatically (without requiring user annotations) and thus subsumes regular string inference.
- We have implemented RIPPLE in SOOT [42], a static analysis and optimization framework for Java and Android programs, with RIPPLE working together with its SPARK, a flow- and context-insensitive pointer analysis. We have evaluated the soundness, precision, scalability and effectiveness of RIPPLE by using 17 popular real-world Android apps from Google Play, in which the data-flows needed for resolving some reflective calls are null. RIPPLE discovers 72 more (true) reflective targets than string inference in seconds at a low false positive rate of 21.9%. This translates into 310 more sensitive data leakages detected by FLOWDROID [5].

The rest of this paper is organized as follows. Section 2 reviews reflection usage and discusses reflection-related security vulnerabilities. Section 3 examines the ubiquity of IIEs. Section 4 describes our RIPPLE approach. Section 5 gives a formalization of RIPPLE. Section 6 evaluates RIPPLE with real-world Android apps. Section 7 discusses the related work. Finally, Section 8 concludes.

2. BACKGROUND

We review reflection usage in Android apps and discuss security issues caused if reflection is not handled well.

2.1 Reflection Usage

Android apps are coded in Java. The Java reflection API provides metaobjects for inspecting classes, methods, and fields at runtime.

In Figure 1, `clz` and `mtd` are metaobjects of classes `Class` and `Method`, respectively. Their names are obtained from `Intent` and `SharedPreferences`, two Android’s built-in containers. In lines 1 – 2, the class name `cName` for `clz` is retrieved from an intent. Subsequently, in line 6, `clz` is created in a call to `Class.forName()`. In line 7, `v` points to an object reflectively created by `clz.newInstance()`. In lines 3 – 4, the method name for `mtd` is retrieved from the map `prefs` as the value associated with the key “`mtd`” or “`foo`” if the key “`mtd`” does not exist yet. In line 8, `mtd` is created in a call to `clz.getMethod()`, which returns a public method declared in or inherited by `clz` with a single parameter of type `A`. Finally, in line 9, this method is called reflectively by `mtd.invoke(v, a)` on the receiver object pointed by `v` with the actual argument being `a`. If `mtd` is static, then `v` is null.

Reflection introduces implicitly the caller-callee edges into the call graph of the program. If one reflective call, e.g., `Class.forName()`, `clz.getMethod()`, `clz.newInstance()` or `mtd.invoke()`, is ignored, the caller-callee edges in line 9 will not be discovered. As a result, possible security vulnerabilities in the invisible part of the program will go undetected.

Therefore, the objective of reflection analysis is to discover the targets at reflective calls (e.g., objects created, methods called and fields accessed), by working with pointer analysis.

2.2 Reflection-Related Security Issues

```

1 Intent intent = activity.getIntent();
2 String cName = intent.getStringExtra("class");
3 SharedPreferences prefs = getSharedPreferences(PrefName, 0);
4 String mName = prefs.getString("mtd", "foo");
5 A a = new A();
6 Class clz = Class.forName(cName);
7 Object v = clz.newInstance();
8 Method mtd = clz.getMethod(mName, A.class);
9 mtd.invoke(v, a);
...

```

Figure 1: An example of reflection usage in Android.

In Android apps, reflection serves a number of purposes, including (1) plug-in and external library support, (2) hidden API method invocation, (3) access to private API methods and fields, and (4) backward compatibility. Indeed, reflection is widely used in both benign and malicious Android apps. For a sample of 202 top-chart free apps from Google Play that we analyzed on 15 April 2016, we found that 92.6% of these apps use reflection. Elsewhere, in a malware sample consisting of 6,141 Android apps from the VriusShare project [1], 48.13% of them also use reflection.

Reflection is responsible for a number of security exploits in Android apps. In general, a malicious app retrieves class and method names as strings externally and invokes methods in payload classes via reflection to perform malicious activities, which are thus disguised from some signature-based anti-virus software. For example, Obad [41] and FakeInstaller [33], represent the two most sophisticated malware families [31], as they combine reflection and code obfuscation to hide their malicious behaviors. In both cases, the methods that are used to collect and steal sensitive data are invoked reflectively with encrypted class and method names. To elude detection by dynamic analyzers such as Google Bouncer [26], malicious behaviors are also suppressed by using logic bombs [12] and emulation detection mechanisms.

As a result, effective static analysis for reflection with a good precision is needed by the analysis community for Android apps. With such a tool, the malicious behaviors in malware and the security vulnerabilities caused by misused reflection in goodwillware can both be detected.

3. IIES IN ANDROID APPS

There are two types of missing information under IIEs. In one case, the data flows needed for resolving reflective calls exist but are statically unknown. Consider the code given in Figure 1. During the static analysis, `cName` and `mName` may be non-null but are statically unknown. Similarly, `v` may point to a non-null object but with a statically unknown type. In this case, we can resolve the reflective calls in lines 6 – 9 by performing type inference to infer what `clz`, `mtd`, and the objects pointed to by `v` are, as done previously for Java programs [18, 19, 37].

In the other case, the data flows needed for resolving reflective calls are completely missing, indicated by the presence of null. To understand this case, which has never been studied before, for Android apps, we have performed an empirical study on 45 Android apps, with 20 popular Android apps from Google Play and 25 malware samples from the VirusShare Project [1]. We discuss the four most common sources of incomplete information in IIEs: (1) undetermined intents, (2) behavior-unknown libraries, (3) unresolved built-in containers, and (4) unmodeled services. We

delve into their bytecode to explain why regular string inference is inadequate, since it fails to enable FLOWDROID [5] to discover many data leaks from *sources* (API calls that inject sensitive information) to *sinks* (API calls that leak information). We also provide insights on why our IIE-aware RIPPLE can handle IIEs more effectively than string inference.

3.1 Undetermined Intents

ICC via intents is one of the most fundamental features in Android as it enables some components to process the data originating from other components. Thus, the components in an Android app function as building blocks for the entire system, enhancing intra- and inter-application code reuse.

In practice, some inter-component control- and data-flows cannot be captured by ICC analysis [27–29]. If a data flow from an intent into a reflective call is missing, the reflection call cannot be fully resolved. The code snippet in Figure 2 taken from the game *Angry Birds* illustrates this problem.

Android App Name: Angry Birds

```
1 public class MMAActivity extends Activity {
2   protected void onCreate(Bundle savedInstanceState) {
3     Intent intent = getIntent();
4     String cName = intent.getStringExtra("class");
5     Class clz = Class.forName(cName);
6     MMBaseActivity mmBase = (MMBaseActivity) clz.newInstance();
7     mmBase.onCreate(savedInstanceState);
8     ... } }
```

Figure 2: Undetermined intents. Here, \dashrightarrow denotes a missed data-flow and \longrightarrow the post-dominating-cast-based type inference used in Ripple. These notations are also used in Figures 3 – 5.

In this app, the class name `cName` is obtained from an intent (line 4) and then used in a call to `Class.forName()` (line 5) to create a class metaobject `clz`. Then, an object of this class is created reflectively (line 6) and assigned to `mmBase` after a downcast to `MMBaseActivity` is performed. Finally, `onCreate()` is invoked on this object (line 7).

To discover what `cName` is, we applied IC3, a state-of-the-art ICC analysis [28], but to no avail. Thus, the data-flow for `cName`, denoted by \dashrightarrow , is missing, rendering `clz` to be a null pointer. In this case, string inference is ineffective. As a result, the reflectively allocated object in line 6 and the subsequent call on this object in line 7 are ignored.

RIPPLE is aware of the incomplete information caused by this undetermined intent, which manifests itself in the form of `cName = null`. By taking advantage of the post-dominant cast `MMBaseActivity` for `clz.newInstance()` in line 6, RIPPLE infers that `mmBase` may point to five objects with their types ranging over `MMBaseActivity` and its four subtypes, which are all confirmed to be possible by manual code inspection. As a result, RIPPLE discovers 3,928 caller-callee edges in lines 5 – 7 (directly or indirectly), thereby enabling FLOWDROID [5] to detect 49 new sensitive data leaks that will be all missed by string inference in this part of the app that has been made analyzable by RIPPLE.

3.2 Behavior-Unknown Libraries

To accelerate the analysis of an application, the side effects of a library on the application are often summarized. Library summaries are either written manually [13] or generated automatically [4, 7]. However, both approaches are

error-prone and often fail to model all the side-effects of a library for all possible analyses. DroidSafe [13] provides the Android Device Implementation (ADI) to model the Android API and runtime manually, with about 1.3 MLOC for Android 4.4.3. However, as the Android framework evolves with both new features and undocumented code added, how to keep this ADI in sync can be a daunting task.

Therefore, unsound library summaries are an important source of incomplete information in IIEs. The code snippet in Figure 3 taken from the app *Twist* illustrates this issue.

Android App Name: Twist

```
1 private static void writeToLog(UnityAdsDeviceLogEntry entry) {
2   String mName = entry.getLogLevel().getReceivingMethodName();
3   Method logMtd = Log.class.getMethod(mName,
4                                     String.class, String.class);
5   String tag = ...;
6   String msg = ...;
7   logMtd.invoke(null, tag, msg);
8 }
9
10 public class Log {
11   public static int i(String tag, String msg) {}
12   public static int d(String tag, String msg) {} Sink
13   public static int w(String tag, String msg) {} Calls
14   public static int e(String tag, String msg) {}
15   public static int v(String tag, String msg) {}
16   public static int wtf(String tag, String msg) {}
17   ... }
```

Figure 3: Behavior-unknown libraries. Here, \dashrightarrow marks the target methods invoked at a reflective call, \longrightarrow denotes sensitive data-flow, and \bullet denotes tainted data. These notations, together with those in Figure 2, are also used in Figures 4 and 5.

This code snippet is used to log messages at different verbosity levels. In line 2, a method name `mName` is retrieved. In line 3, its method metaobject `logMtd` is created. In line 6, this method, which is static, is invoked reflectively.

If we apply FLOWDROID [5] to detect data leaks in this app, by relying on string inference to perform reflection analysis, then the reflective call `logMtd.invoke()` in line 6 will be ignored. In FLOWDROID, the behaviors of maps are not summarized. However, `entry` was retrieved from a `HashMap` and then passed to `writeToLog`. Thus, `mName = null`, rendering string inference to be ineffective.

RIPPLE is aware of unsound library summaries and thus attempts to infer the target methods at `logMtd.invoke()`. Based on the facts that (1) these methods are static (since the receiver object is null), declared in or inherited by class `android.util.Log`, (2) each target method has two formal parameters, and (3) each parameter has a type that is either `String` or its supertype or its subtype, RIPPLE concludes that the six target methods, `i()`, `d()`, `w()`, `e()`, `v()` and `wtf()`, as shown in class `android.util.Log` may be potentially invoked. According to FLOWDROID, these six methods are all sinks for sensitive data contained in `msg`. Thus, resolving `logMtd.invoke()` causes 12 data leaks from two different sensitive data sources to be reported (as 2 sources \times 6 sinks = 12 leaks). By manual code inspection, we found that the first four methods, `i()`, `d()`, `w()` and `e()`, shaded in class `android.util.Log` are true targets, implying that 8 data leaks will not be reported if string inference is used.

3.3 Unresolved Built-in Containers

Android apps can receive a variety of user inputs from, e.g., intents, databases, internet, GUI actions, and system

events. These data are stored in different types of containers, such as **Bundle**, **SharedPreferences**, **ContentValues** and **JSONObject**, for different purposes. Unhandled user inputs represent an important source of incomplete information in IIEs. The code snippet in Figure 4 taken from a game named *Seven Knights* illustrates this problem.

Android App Name: Seven Knights

```

1 public static WXMediaMessage fromBundle(Bundle bundle) {
2   String cName = bundle.getString("_wxobject_identifier_");
3   Class clz = Class.forName(cName);
4   IMediaObject media = (IMediaObject) clz.newInstance();
5   media.unserialize(bundle); }
6 public class WXFileObject implements IMediaObject {
7   public void unserialize(Bundle bundle) {
8     Object s = bundle.getString("_wxfileobject_filePath");
9     ... } }

```

Figure 4: Unresolved Bundles.

In this code snippet, different types of objects are created (line 4) to handle different types of media data according to their unique identifiers stored in a **Bundle** (line 2), which is constructed according to the types of media introduced by third-party apps. **IMediaObject** is an interface implemented by eight types (i.e., classes) of media, with only **WXFileObject** shown partially. Therefore, **cName** represents the name of one of these eight classes. In line 4, **media** points to a reflectively created object of the class identified by **cName**. In line 5, a call is made to **unserialize()** on the receiver object pointed to by **media** with **bundle** as its argument.

If we apply again FLOWDROID to detect data leaks in this app, by relying on string inference to resolve the reflective calls in the app, then **cName** will be null, since the behaviors of bundles are not modeled in FLOWDROID. As a result, the reflectively created object in line 4 and the subsequent call on this object in line 5 will be ignored.

By being IIE-aware, RIPPLE will infer the inputs retrieved from **bundle** to resolve the call to **clz.newInstance()** in line 4. By taking advantage of the post-dominant cast **IMediaObject** for this reflective call, RIPPLE deduces that **media** points to potentially eight objects with their types ranging over all the eight classes implementing **IMediaObject**, confirmed by manual code inspection. As a result, a total of 37 caller-callee edges, together with 16 sensitive data sources, which would otherwise be missed by string inference, are discovered in lines 3 – 5 directly or indirectly. Currently, these 16 sensitive data sources do not flow to any sinks but may do so in a future app release. The resulting leaks will be then detected by FLOWDROID, assisted by RIPPLE.

3.4 Unmodeled Services

The Android framework provides an abstraction of abundant services for a mobile device, such as obtaining the device status, making phone calls, and sending text messages, which are all related to critical program behaviors. These services are usually initialized during system startup and subsequently used by calling the factory methods in the Android framework with often reflective calls involved. Unsound modeling for Android’s system-wide services can be an important source of incomplete information in IIEs. The code snippet in Figure 5 taken from a text message management app named *GO SMS Pro* illustrates this issue.

In line 2, **clz** represents a class metaobject for **android**.

Android App Name: Go SMS Pro

```

1 public void getSubscriberId() {
2   Class clz = Class.forName("android.telephony.TelephonyManager");
3   Method getDefMtd = clz.getMethod("getDefault");
4   Object telephonyManager = getDefMtd.invoke(null);
5   Method getSubIdMtd = clz.getMethod("getSubscriberId");
6   String id = (String) getSubIdMtd.invoke(telephonyManager);
7   ... }

```




Figure 5: Unmodeled services.

telephony.TelephonyManager. In line 3, **getDefMtd** represents a method metaobject for a static method named **getDefault** in **clz**. In line 4, this method is invoked reflectively, with its returned object, an instance of **clz**, assigned to **telephonyManager**. In lines 5 – 6, a method metaobject, **getSubIdMtd**, for an instance method named **getSubscriberId** in **clz** is created and then invoked reflectively on the receiver object pointed to by **telephonyManager**.

In this code snippet, all the class and method names are string constants. Thus, regular string inference can resolve precisely the reflective targets at all the reflective calls shown. However, this still does not enable the target methods invoked in line 6 to be analyzed, because the **getDefault** method invoked in line 4 is part of the hidden API and thus not available for analysis. Thus, **telephonyManager** is null, causing the reflective call in line 6 to be skipped.

RIPPLE is aware of the existence of unmodeled services. By examining the class type in the **getSubIdMtd** metaobject, RIPPLE concludes that **telephonyManager** points to an object of type **android.telephony.TelephonyManager**. As a result, the reflective call in line 6 can be resolved, resulting in the target method **getSubscriberId** to be discovered. For this app, FLOWDROID is unscalable. Otherwise, the potential data link as shown will be detected automatically.

Finally, if the **getDefault** method is native with its method body unmodeled but available for analysis, then RIPPLE will be able to infer in line 4 that **telephonyManager** may point to an object of type **android.telephony.TelephonyManager**. As a result, the reflective call in line 6 can also be resolved.

4. METHODOLOGY

Figure 6 depicts an overview of RIPPLE, an IIE-aware reflection analysis introduced in this paper for Android apps. Currently, we consider four important contributing factors to IIEs when the data-flows needed for resolving reflective calls are null: undetermined intents, behavior-unknown libraries, unresolved containers and unmodeled services, as discussed in Section 3. To handle IIEs effectively, RIPPLE resolves reflective calls in the presence of incomplete information about these calls, so that their induced caller-callee edges can be discovered. Once the call graph of an app is available, many client analyses, such as data leakage detection, security policy verification, malware detection, and security vetting, can be performed to detect various security issues, especially those hidden in some reflective calls, in the app.

We have developed RIPPLE by leveraging recent advances on reflection analysis for Java [18, 19]. Conceptually, RIPPLE performs reflection analysis by distinguishing three cases:

- **Case 1. String Inference for Constant Strings.** If class/method/field names used at reflective calls are string

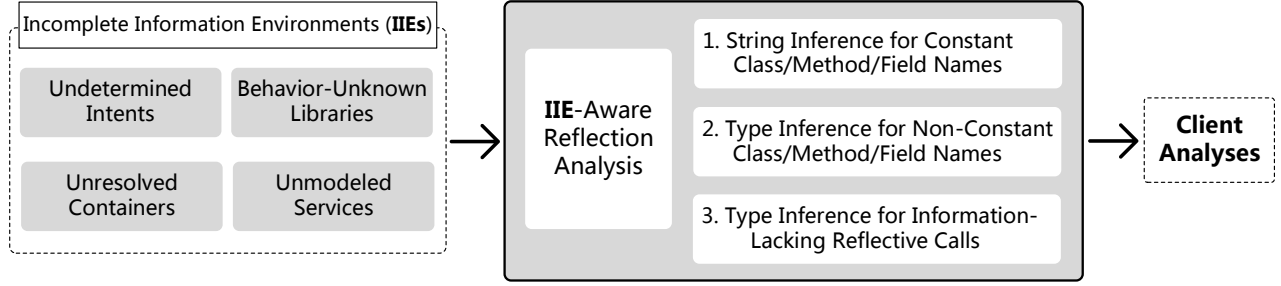


Figure 6: An overview of RIPPLe.

constants, regular string inference is conducted.

- **Case 2. Type Inference for Unknown Strings.** If class/method/field names are non-constant but non-null strings, which may be read from configuration files or command lines, then type inference that was previously introduced for Java programs [18, 19, 21, 37] can be leveraged.
- **Case 3. Type Inference for Information-Missing Reflective Calls.** Again, type inference is performed to infer the missing information at reflective calls in the following three categories, as reviewed in Section 3:
 - **Null-Name.** Class, method or field names are null, as illustrated in Figure 2 (with `cName = null` for undetermined intents), Figure 3 (with `mName = null` for behavior-unknown libraries), and Figure 4 (with `cName = null` for unmodeled bundles). We will replace a null object by an unknown string and then go back to Case 2.
 - **Missing-RecvObj.** Given a call to `mtd.invoke(y, ...)`, a target method pointed to by `mtd` does not have a corresponding receiver object pointed to by `y`, as illustrated in Figure 5 for the `getSubscriberId` method invoked in line 6, where `telephonyManager = null`. If we know the class type of the target method pointed to by `mtd`, its corresponding receiver object can be inferred.
 - **Missing-RetObj.** Given `x = mtd.invoke(...)`, a target method pointed to by `mtd` is available for analysis but its method body is unmodeled. This can happen to `telephonyManager = getDefMt.invoke(null)` in Figure 5, as discussed in Section 3, when the `getDefault` method is hypothetically assumed to be an unmodeled native method. In general, if we know the return type of the target method pointed to by `mtd`, then objects of this type or its subtypes are created and assigned to `x`.

To the best of our knowledge, RIPPLe is the first automated reflection analysis (without relying on user annotations) for handling Cases 2 and 3 and also the first for handling Cases 1 – 3 in a unified framework for Android apps.

In RIPPLe, reflection analysis is performed together with pointer analysis mutually recursively, as effectively one single analysis. On one hand, reflection analysis makes use of the points-to information to resolve reflective calls. On the other hand, pointer analysis needs the results of reflection analysis to determine which methods get called and which fields get accessed. Once the entire analysis for an app is over, its call graph is readily available (at the same time).

5. FORMALISM

We formalize RIPPLe as a form of reflection analysis, performed together with a flow- and context-insensitive pointer analysis. We restrict ourselves to a small core of the Java reflection API, including `Class.forName()`, `Class.newInstance()`, `Class.getMethod()`, and `Method.invoke()`. For simplicity, we consider only instance methods as static methods are handled similarly. We focus only on reflective method calls as our formalism extends easily to reflective field accesses.

5.1 Notations

Figure 7 gives the domains used in our formalism. The abstract heap objects are labeled by their allocation sites. \mathbb{C} represents the set of class metaobjects and \mathbb{M} the set of method metaobjects. The class type of a class or method metaobject is identified by its superscript and the signature of a method metaobject, which consists of the method name and descriptor (i.e., return type and parameter types), is identified by its subscript. In particular, u indicates an unknown class type or an unknown method signature (with some parts of the signature being statically unknown).

class type	$t, u \in \mathbb{T}$
variable	$v \in \mathbb{V}$
abstract heap object	$o_1^t, o_2^t, o_3^t, \dots \in \mathbb{O}$
class metaobject	$c^t, c^u, c^-, \dots \in \mathbb{C}$
method metaobject	$m_s^t, m_s^u, m_t^u, m_u^u, m_-, \dots \in \mathbb{M} = \mathbb{T} \times \mathbb{S}$
method name	$n \in \mathbb{N}$
method parameter type	$p \in \mathbb{P} = \bigcup_{i=0}^{\infty} \mathbb{T}^i$
method signature	$s, u \in \mathbb{S} = \mathbb{T} \times \mathbb{N} \times \mathbb{P}$

Figure 7: Domains.

5.2 Pointer Analysis

Figure 8 gives a standard formulation of a flow- and context-insensitive Andersen’s pointer analysis [2]. $pts(v)$ represents the points-to set of a pointer v . An array object is analyzed with its elements collapsed to a single field, say, *arr*.

In a reflection-free program, only five types of statements exist. In [P-NEW], o_i^t uniquely identifies the abstract object created as an instance of t at this allocation site, labeled by i . In [P-COPY], the points-to facts flow from the RHS to the LHS of a copy statement. In [P-LOAD] and [P-STORE], the fields of an abstract object o_i^t are distinguished.

In [P-CALL] (for non-reflective calls), the function $dispatch(o_i^t, m)$ is used to resolve the virtual dispatch of method m on the receiver object o_i^t to be m' . We assume that m' has a formal parameter m'_{this} for the receiver object and $m'_{p_0}, \dots, m'_{p_{n-1}}$ for the remaining parameters, and a pseudo-variable m'_{ret} is used to hold the return value of m' .

$i : x = \text{new } t() \quad [\text{P-NEW}]$	$x = y \quad [\text{P-COPY}]$
$\frac{\{o_i^t\} \subseteq pts(x)}{x = y.f \quad [\text{P-LOAD}]}$	$\frac{pts(y) \subseteq pts(x)}{x.f = y \quad [\text{P-STORE}]}$
$\frac{o_i^t \in pts(y)}{pts(o_i^t.f) \subseteq pts(x)}$	$\frac{o_i^t \in pts(x)}{pts(y) \subseteq pts(o_i^t.f)}$
$x = y.m(arg_0, \dots, arg_{n-1}) \quad [\text{P-CALL}]$	
$\frac{o_i^t \in pts(y) \quad m' = \text{dispatch}(o_i^t, m)}{\{o_i^t\} \subseteq pts(m'_{\text{this}}) \quad pts(m'_{\text{ret}}) \subseteq pts(x) \quad \forall 0 \leq k < n : pts(arg_k) \subseteq pts(m_{p_k})}$	

Figure 8: Rules for pointer analysis.

5.3 Reflection Analysis: Cases 1 and 2

Figure 9 gives the rules for resolving reflective calls for Cases 1 and 2 in Figure 6 simultaneously. RIPPLE is the first to combine both cases in reflection analysis for Android apps. In Case 1, regular string inference for constant class and method names is applied. In Case 2, type inference for non-constant but non-null class and method names is applied. As discussed earlier, reflective field accesses are omitted. Recall that $<$ denotes the subtyping relation.

Let us now examine the five rules for Cases 1 and 2. [C12-FORNAME] handles a `Class.forName(cName)` call. For the auxiliary function $toClass : \mathbb{O} \rightarrow \mathbb{C}$, $toClass(o_{\text{String}}^{\text{String}})$ takes a string object $o_{\text{String}}^{\text{String}}$ and returns its corresponding class metaobject. If $o_{\text{String}}^{\text{String}}$ is a constant, then $toClass(o) = c^t$, where t is the class type named by `cName`. Otherwise, $toClass(o_{\text{String}}^{\text{String}}) = c^u$, since `cName` is a non-constant but non-null string. In this rule, `clz` is thus made to point to either c^t or c^u accordingly. In the case of c^u , the missing type may be inferred when it is used in a subsequent reflective call.

[C12-GETMTD] handles a `clz.getMethod(mName, ...)` call analogously. For the auxiliary function, $toMtdSig : \mathbb{C} \times \mathbb{O} \rightarrow \mathcal{P}(\mathbb{S})$, $toMtdSig(c^-, o_{\text{String}}^{\text{String}})$ returns the set of method signatures for the methods declared in or inherited by the class c^- with their method name identified by `mName`. If $c^- = c^u$ but `mName` is a string constant, say, “foo”, then `foo` is recorded: $toMtdSig(c^-, o_{\text{String}}^{\text{String}}) = \{(u, \text{foo}, u)\}$. If `mName` is a non-constant but non-null string, then $toMtdSig(c^-, o_{\text{String}}^{\text{String}}) = \{(u)\}$. Therefore, this rule distinguishes two cases for every signature $s \in toMtdSig(c^-, o_{\text{String}}^{\text{String}})$. If $c^- = c^t$ is a statically known type t , a method metaobject m_s^t is created. Otherwise, a method metaobject m_s^u is created. In both cases, the missing information in a method metaobject may be inferred when it is used in a subsequent reflective call.

[C12-NEW] handles reflective object allocation at a call to `x = (T) clz.newInstance()`, where `T` symbolizes an intra-procedurally post-dominating type cast for the call if it exists or `java.lang.Object` otherwise. If $c^- = c^t$ is a statically known type t , then `x = clz.newInstance()` degenerates into `x = new t()` and can thus be handled as in [P-NEW]. Otherwise, $c^- = c^u$. If $T \neq \text{java.lang.Object}$, then u is inferred to be `T` or any of its subtypes.

There are two rules for handling reflective method invocation. To infer a target method invoked reflectively, we need to infer its class type, which is handled by [C12-INVTYPE], and its signature, which is handled by [C12-INVSIG].

[C12-INVTYPE] is simple. The class type of a method metaobject m_s^u is inferred to be m_s^t for every possible dy-

$\text{Class clz} = \text{Class.forName}(cName) \quad [\text{C12-FORNAME}]$
$\frac{o_{\text{String}}^{\text{String}} \in pts(cName) \quad c^- = toClass(o_{\text{String}}^{\text{String}})}{pts(clz) \supseteq \begin{cases} \{c^t\} & \text{if } c^- = c^t \\ \{c^u\} & \text{if } c^- = c^u \end{cases}}$
$\text{Method mtd} = clz.getMethod(mName, \dots) \quad [\text{C12-GETMTD}]$
$\frac{o_{\text{String}}^{\text{String}} \in pts(mName) \quad c^- \in pts(clz) \quad s \in toMtdSig(c^-, o_{\text{String}}^{\text{String}})}{pts(mtd) \supseteq \begin{cases} \{m_s^t\} & \text{if } c^- = c^t \\ \{m_s^u\} & \text{if } c^- = c^u \end{cases}}$
$i : x = (T) clz.newInstance() \quad [\text{C12-NEW}]$
$\frac{c^- \in pts(clz)}{pts(x) \supseteq \begin{cases} \{o_i^t\} & \text{if } c^- = c^t \\ \{o_i^t \mid t <: T\} & \text{if } c^- = c^u \end{cases}}$
$_ = mtd.invoke(y, \dots) \quad [\text{C12-INVTYPE}]$
$\frac{o_{\text{String}}^{\text{String}} \in pts(y) \quad m_s^u \in pts(mtd)}{pts(mtd) \supseteq \{m_s^t\}}$
$_ = (T) mtd.invoke(_, args) \quad [\text{C12-INVSIG}]$
$\frac{m_s^u \in pts(mtd) \quad t \ll: T \quad p \in toParaTys(args)}{pts(mtd) \supseteq \{m_s^- \mid s.\text{para} = p \wedge s.\text{ret} = t\}}$

Figure 9: Rules for Cases 1 and 2 in Figure 6.

namic type t of every receiver object pointed to by y .

[C12-INVTYPE] is slightly more involved in handling `(T) mtd.invoke(_, args)`, where `T` is defined identically as in [C12-NEW]. This rule attempts to infer the missing information in the signature s of a method from its `args` and its possible return type. Here, we write `s.para` and `s.ret` to identify its parameter types and return type, respectively. The typing relation \ll is defined by distinguishing two cases. First, $u \ll: \text{java.lang.Object}$. Second, if t is not `java.lang.Object`, then $t' \ll: t$ holds if and only if $t' <: t$ or $t <: t'$ holds. Therefore, `s.ret` is deduced from a post-dominating cast `T` (which is not `java.lang.Object`). As for `s.para`, we infer it intra-procedurally from `args`. For the auxiliary function $toParaTys : \bigcup_{i=0}^{\infty} \mathbb{V}^i \rightarrow \mathcal{P}(\mathbb{S})$, $toParaTys(args)$ returns the set of parameter types of a target method invoked with its argument list `args`, computed only intra-procedurally for efficiency reasons. If `args` is not defined locally, i.e., not in the same method containing the reflective method call, then $toParaTys(args) = \emptyset$. Otherwise, let D_i be the set of declared types of all possible variables assigned to the i -th argument `args[i]`. Let $P_i = \{t' \mid t \in D_i \wedge (t' <: t \vee t <: t')\}$. Then, $toParaTys(args) = P_0 \times \dots \times P_{n-1}$. With `s.para` or `s.ret` inferred for m_s^- , `mtd` is made to point to a new method metaobject m_s^- , where s contains the missing information in u deduced via inference.

5.4 Reflection Analysis: Case 3

Figure 10 gives the rules for resolving reflective calls for Case 3 in Figure 6. There are four rules for handling three categories of incomplete information, NULL-NAME, MISSING-RECVOBJ and MISSING-RETOBJ, as discussed in Section 4.

[C3-FORNAME] and [C3-GETMTD] deal with NULL-NAME by treating null as a non-constant string and then resorting to [C12-NEW], [C12-INVTYPE] and [C12-INVSIG] to infer the missing information. [C3-FORNAME] handles a `Class.forName(cName)` call with `cName` = null identically as how [C12-FORNAME] handles a `Class.forName(cName)` call when `cName` is a non-constant string. Similarly, [C3-GETMTD]

Class $clz = \text{Class.forName}(cName)$	[C3-FORNAME]
$\frac{pts(cName) = \emptyset}{pts(clz) \supseteq \{c^u\}}$	
Method $mtd = clz.getMethod(mName, _)$	[C3-GETMTD]
$\frac{pts(mName) = \emptyset \quad c \in pts(clz)}{pts(mtd) \supseteq \begin{cases} \{m_u^t\} & \text{if } c = c^t \\ \{m_u^u\} & \text{if } c = c^u \end{cases}}$	
$i : _ = mtd.invoke(y, _)$	[C3-INVRECV]
$\frac{t'' \in (\{t \mid m_- \in pts(mtd)\} \setminus \{t' \mid o_-^t \in pts(y) \wedge t <: t'\}) \quad t'' \neq \text{java.lang.Object}}{pts(y) \supseteq \{o_i^{t''} \mid t''' \ll: t''\}}$	
$i : x = mtd.invoke(_, _)$	[C3-INVRET]
$\frac{m_s^- \in pts(mtd) \quad s.ret = t \quad t' <: t \quad \forall o_-^{t''} \in pts(x) : t'' \not\prec: t \quad t \neq \text{java.lang.Object}}{pts(x) \supseteq \{o_i^{t'}\}}$	

Figure 10: Rules for Case 3 in Figure 6.

handles a `clz.getMethod(mName, ...)` call with `mName = null` identically as how [C12-GETMTD] handles a `clz.getMethod(mName, ...)` call when `mName` is a non-constant string.

[C3-INVRECV] handles MISSING-RECVOBJ by inferring the missing receiver objects pointed to by `y` from the known class types of all possibly invocable target methods, except that `java.lang.Object` is excluded for precision reasons. This rule covers an important special case when $pts(y) = \emptyset$.

[C3-INVRET] handles MISSING-RETOBJ by inferring the missing objects returned from a target method that is unmodeled (with its body missing) but available for analysis from the return type `s.ret` of its signature `s`. Objects of all possible subtypes of `s.ret` are included in $pts(x)$, unless `x` already points to an object of one of these subtypes.

5.5 Transforming Reflective to Regular Calls

Fig. 11 shows how to transform a reflective into a regular call, which will be analyzed by pointer analysis.

$x = mtd.invoke(y, args)$	[T-INV]
$\frac{m_s^- \in pts(mtd) \quad m' \in MTD(m_s^-) \quad o_i^- \in pts(args) \quad o_j' \in pts(o_i^- . arr) \quad t'' \text{ is declaring type of } m'_{pk} \quad k \in [0, n-1] \quad t' <: t''}{\{o_j^{t'}\} \subseteq pts(arg_k) \quad x = y.m'(\arg_0, \dots, \arg_{n-1})}$	

Figure 11: Rule for Transformation.

For the auxiliary function $MTD : \mathbb{M} \rightarrow \mathcal{P}(M)$, where M is the set of methods in the program, $MTD(m_s^-)$ is the standard lookup function for finding the methods in M according to a declaring class t and a signature s for a method metaobject, except that (1) the return type in s is also considered and (2) any u in s is treated as a wild card.

As discussed earlier, `args` points to a 1-D array of type `Object[]`, with its elements collapsed to a single field `arr` during the pointer analysis. Let `arg0, ..., argn-1` be the n freshly created arguments to be passed to each potential target method m' found by in $MTD(m_s^-)$. Let $m'_{p_0}, \dots, m'_{p_{n-1}}$ be the n parameters (excluding `this`) of m' , such that the declaring type of m'_{pk} is t'' . We include $o_j^{t'}$ to $pts(arg_k)$ only when $t' <: t''$ holds in order to filter out the objects that

cannot be assigned to m'_{pk} . Finally, the regular call obtained can be analyzed by [P-CALL] in Figure 8.

5.6 Examples

Let us apply RIPPLE to the app in Figure 2. Due to an undetermined intent in line 3, `cName` is null in line 4. By applying [C3-FORNAME] to `Class.forName(cName)` in line 5, we obtain $pts(clz) = \{c^u\}$. In line 6, we apply [C12-NEW] to obtain $pts(mmBase) = \{o_6^{MMBaseActivity}, o_6^{AdViewOverlayActivity}, o_6^{BridgeMMMedia$PickerActivity}, o_6^{CachedVideoPlayerActivity}, o_6^{VideoPlayerActivity}\}$, as these five class types are deduced from the post-dominating cast `MMBaseActivity` for the call to `clz.newInstance()` in line 6. As a result, RIPPLE discovers 3,928 caller-callee edges in lines 5 – 7 (directly or indirectly), thereby enabling FLOWDROID [5] to detect 49 sensitive data leaks along these edges.

Let us apply RIPPLE to the app in Figure 5. As `getDefault` is a hidden API method and thus unavailable for analysis, `telephonyManager = null`. By applying [C3-INVRECV] to `getSubIdMtd.invoke(telephonyManager)` in line 6, we can deduce precisely that $pts(telephonyManager) = \{o_6^{TelephonyManager}\}$. In this app, `TelephonyManager` has no subtypes but only `java.lang.Object` as its supertype. This allows the potential leak highlighted in Figure 5 to be detected. If `getDefault` were unmodeled but available for analysis, then we would be able to apply [C3-INVRET] to `getDefMtd.invoke(null)` in line 4 to infer the missing object returned: $pts(telephonyManager) = \{o_4^{TelephonyManager}\}$. Again, the same sensitive data leak will be detected.

6. EVALUATION

We demonstrate that RIPPLE is significantly more effective than string inference for real-world Android apps by addressing the following four research questions:

- **RQ1.** Is RIPPLE capable of discovering more reflective targets, i.e., more sound than string inference?
- **RQ2.** Can RIPPLE achieve this with a good precision?
- **RQ3.** Does RIPPLE scale for real-world Android apps?
- **RQ4.** Is RIPPLE effective in enabling existing Android security analyses to detect security vulnerabilities?

To answer RQ1 – RQ3, we examine the reflective targets resolved by RIPPLE in Android apps. To answer RQ4, we investigate how RIPPLE enables FLOWDROID [5], a taint analysis for Android apps, to find more sensitive data leaks.

Android Apps. Real-world Android apps rather than synthetic benchmarks are used in our experiments. We examined the top-chart free applications from Google Play downloaded on 15 April 2016, which are the most popular apps in the official app store. A set of 17 apps is selected, such that they exhibit a wide range of incomplete information, with null class and method names, as discussed in Section 3, and they are scalable under FLOWDROID within 2 hours.

State-of-the-Art Reflection Analysis. To resolve reflection for Android apps, there are only two existing static techniques, with both performing string inference (for constant strings), CHECKER [6, 10] and DROIDRA [17]. We cannot compare with CHECKER since its reflection analysis relies on user annotations. We cannot compare with DROIDRA either, since its latest open-source tool (released on 9 September 2016) is unstable. For the 17 apps selected, DROIDRA

Table 1: Soundness and precision. RIPPLE always finds every true reflective target that STRINF does. The numbers in bold indicate that RIPPLE finds more true targets than STRINF. A reflective call in app is considered to be reachable if it can be reached from the harness `main()` in its call graph.

App (Package Name)	STRINF								RIPPLE							
	Class.newInstance()				Method.invoke()				Class.newInstance()				Method.invoke()			
	#Calls		#Targets		#Calls		#Targets		#Calls		#Targets		#Calls		#Targets	
	Reachable	Resolved	Resolved	True	Reachable	Resolved	Resolved	True	Reachable	Resolved	Resolved	True	Reachable	Resolved	Resolved	True
com.facebook.orca	0	0	0	0	7	0	0	0	0	0	0	0	7	1	7	7
com.netmarble.sknightsgb	2	0	0	0	11	4	8	4	2	2	26	11	11	4	8	4
com.productmadness.hovmobile	1	0	0	0	7	5	31	29	1	1	11	1	7	5	31	29
com.facebook.moments	0	0	0	0	5	0	0	0	0	0	0	0	5	1	7	7
me.msgrd.android	0	0	0	0	11	4	4	4	0	0	0	0	11	4	4	4
com.nordecurrent.canteenhd	3	0	0	0	16	5	6	5	4	3	27	10	20	5	6	5
com.es.game.sincitymobile_row	0	0	0	0	6	2	2	2	0	0	0	0	6	2	2	2
com.imangi.templerun	0	0	0	0	7	1	1	1	0	0	0	0	7	1	1	1
com.rovio.angrybirds	3	1	1	1	10	3	3	3	3	2	6	6	14	8	13	11
com.sgn.pandapop_gp	0	0	0	0	13	3	3	3	0	0	0	0	13	3	3	3
com.gameloft.android.ANMP.GloftA8HM	1	1	16	16	9	0	0	0	1	1	16	16	9	0	0	0
com.appsorama.kleptocats	2	2	5	5	3	0	0	0	2	2	5	5	3	1	6	4
air.au.com.metro.DumbWaysToDie	1	0	0	0	18	8	34	34	1	1	5	5	21	11	37	37
com.ketchapp.twist	3	2	5	5	8	2	2	2	3	2	5	5	8	3	8	6
com.stupeflix.legend	4	2	2	2	1	0	0	0	4	3	13	5	1	0	0	0
com.maxgames.stickwarlegacy	1	0	0	0	2	1	1	1	1	0	0	0	2	2	7	5
air.com.tutotoons.app	0	0	0	0	14	7	43	43	0	0	0	0	14	7	43	43
animalhairsalon2jungle.free																
Total	21	8	29	29	148	45	138	131	22	17	114	64	159	58	183	168

resolves some reflective targets for 6 apps, fails to produce any outputs for another 6 apps, and crashes for the remaining 5 apps. These problems were reported to and confirmed by the authors, but they still remain despite a recent release.

Both CHECKER and DROIDRA perform essentially regular string inference, which is subsumed by RIPPLE as shown in Case 1 (for constant strings) in Figure 6. CHECKER handles non-constant strings with user annotations only. Instead of comparing with CHECKER and DROIDRA directly, we compare RIPPLE with STRINF, which is a simplified RIPPLE that performs only regular string inference in Case 1.

Implementation. We have implemented RIPPLE in SOOT [42], a static analysis framework for Android and Java programs. RIPPLE works with its SPARK, a flow- and context-insensitive pointer analysis, to resolve reflection and points-to information in a program. Based on the results of this joint analysis, the call graph of the program, on which many security analyses such as FLOWDROID operate, can be constructed.

Currently, RIPPLE is implemented in Java with about 3,500 LOC, handling the most significant reflective calls that affect the static analysis of Android apps: `Class.forName()`, `Class.newInstance()`, `Method.invoke()`, and all four method-introspecting calls, `Method.getMethod()`, `Method.getDeclaredMethod()`, `Method.getMethods()`, and `Method.getDeclaredMethods()`.

Computing Platform. Our experiments are carried out on a Xeon E5-2650 2GHz machine with 64GB RAM running Ubuntu 14.04 LTS. The time measured for analyzing an app by a particular analysis is the average of 20 runs.

6.1 RQ1: More Soundness

Table 1 compares RIPPLE and STRINF in terms of the number of reflective targets discovered at all reflective calls to `Class.newInstance()` and `Method.invoke()`. For the former, a target means a reflectively created object. For the latter, a target means a reflectively invoked method. Each app is uniquely identified by its package name. For each of these two methods, only its calls reachable from the harness `main()` used during the analysis are included. We determine whether a target is true or not by manual code inspection.

By design, RIPPLE always finds every true target that STRINF does. In 11 out of the 17 apps, RIPPLE has suc-

cessfully discovered more true targets than STRINF. This highlights the importance of making reflection analysis fully IIE-aware for Android apps, by handling not only Case 2 as for Java programs [18, 19, 21, 37] but also Case 3.

In the 17 Android apps, RIPPLE finds a total of 64 and 168 but STRINF finds only a total of 29 and 131 true targets for `Class.newInstance()` and `Method.invoke()`, respectively. Therefore, for both methods combined, RIPPLE finds 232 but STRINF finds only 160 true targets in total, yielding a net gain of 72 true targets and thus a 45% increase in soundness on reflection analysis.

Let us revisit some examples in Section 3. Consider the app in Figure 2. For the call to `clz.newInstance()` in line 6, RIPPLE infers five reflectively created objects, which are all true targets configured to provide different forms of advertisement. A similar pattern appears also in the app named *Dumb Ways to Die*. Consider now the app in Figure 4. For the call to `clz.newInstance()` in line 4, RIPPLE infers 8 reflectively created objects, which are all true targets used for handling eight different types of media according to user inputs. All these targets are missed by STRINF, which relies only on a simple string analysis for string constants.

6.2 RQ2: Precision

Table 1 reveals also the false positive rates for both RIPPLE and STRINF. RIPPLE finds a total of 297 reflective targets with 232 true targets, representing a false positive rate of 21.9%. STRINF finds a total of 167 reflective targets with 160 true targets, representing a false positive rate of 4.2%.

Due to 72 more true targets discovered, as discussed in Section 6.1, RIPPLE is regarded to exhibit a satisfactory precision for Android apps. For many security analyses such as security vetting and malware detection, and even debugging, it is important to analyze reflection-related program behaviors even if doing so may cause some false warnings to be triggered. Consider the app in Figure 3. For the call to `logMtd.invoke(null, tag, msg)` in line 6, RIPPLE infers its target to be the six methods, `i()`, `d()`, `w()`, `e()`, `v()` and `wtf()` in class `Log`, where the last two are false positives, enabling FLOWDROID to report 12 leaks via `msg` from two data sources, with 4 from `v()` and `wtf()` being false positives.

We can lift the precision of RIPPLE by improving the precision of its collaborating analyses. Currently, RIPPLE works with SPARK, a flow- and context-insensitive pointer analysis,

Table 2: Efficiency and effectiveness. For efficiency, the analysis times of RIPPLe, STRINF and SPARK are compared, by using the final harness (iteratively) constructed for an app by FlowDroid. For each analysis, the number of call graph (CG) edges discovered is also given. For effectiveness, the number of data leaks, together with sensitive source and sink calls, found by FlowDroid under RIPPLe, STRINF and SPARK are compared. The numbers in bold indicate that FlowDroid reports more leaks under RIPPLe than STRINF.

App Package Name	SPARK					STRINF					RIPPLe				
	CG Edges	Analysis Time (s)	Source Calls	Sink Calls	Total Leaks	CG Edges	Analysis Time (s)	Source Calls	Sink Calls	Total Leaks	CG Edges	Analysis Time (s)	Source Calls	Sink Calls	Total Leaks
com.facebook.orca	5583	2.0	43	115	15	5598	2.1	43	115	15	5605	2.2	43	115	15
com.netmarble.knightsgb	12113	7.9	194	208	92	12148	8.0	194	208	96	12779	8.4	210	212	142
com.productmadness.hovmobile	6009	2.7	119	113	41	6278	3.1	119	113	44	6480	3.2	119	116	48
com.facebook.moments	6632	2.5	39	145	10	6647	2.6	39	145	10	6654	2.7	39	145	10
me.msqr.android	10561	4.7	127	107	24	11064	4.8	127	107	25	11064	4.9	127	107	25
com.nordcurrent.canteenhd	16698	9.8	182	305	182	16759	10	182	305	184	21625	12.8	199	327	289
com.ea.game.simcitymobile.row	8369	4.1	111	170	41	8403	4.2	111	172	50	8403	4.4	111	172	50
com.imangi.templerun	10584	2.3	184	110	54	10592	2.5	184	110	54	10592	2.6	184	110	54
com.rovio.angrybirds	12705	7.0	244	180	66	13448	7.4	245	206	66	17384	10.2	276	266	120
com.sgn.pandapop.gp	10557	5.3	217	365	552	10588	5.9	217	365	575	10590	5.9	217	365	575
com.gameloft.android.ANMP.GloftA8HM	15532	7.0	274	181	126	16015	7.2	285	204	144	16016	6.6	285	204	144
com.apporama.kleptocats	3659	2.7	101	106	39	3707	2.8	101	106	39	3714	3.3	101	112	39
air.au.com.metro.DumbWaysToDie	10059	4.9	204	151	23	10312	5.3	204	151	26	11679	5.9	228	167	103
com.ketchapp.twist	14092	8.7	210	179	93	14144	8.7	210	179	97	14151	8.7	210	185	109
com.stupefix.legend	8803	3.1	97	127	46	8852	3.4	97	127	47	9066	3.5	97	127	47
com.masgames.stickwarlegacy	3829	2.6	76	26	5	3831	2.5	76	26	5	3844	2.4	76	32	17
air.com.tutotoons.app	11788	5.2	86	43	9	12075	5.5	86	43	9	12075	5.7	86	43	9
animalhairsalon2jungle.free															
Total	167573	82.5	2508	2631	1418	170461	86	2520	2682	1486	181721	93.4	2608	2805	1796

in SOOT. RIPPLe will be more precise if a more precise, say, an object-sensitive pointer analysis [24] is used, instead.

In addition, RIPPLe will also be more precise if it works simultaneously with other static analyses, such as intent analysis, for Android apps. The code snippet in Figure 12 taken from *Seven Knights* illustrates this proposition.

Android App Name: Seven Knight

```

1 public class ReferralReceiver extends BroadcastReceiver {
2   private void sendOtherBroadcastReceiver(Context ctx, Intent intent) {
3     Intent criterion = new Intent("com.android.vending.INSTALL_REFERRER");
4     List<ResolveInfo> infoList = ctx.getPackageManager()
5       .queryBroadcastReceivers(criterion, 0);
6     for(ResolveInfo info : infoList)
7       if(! getClass().getName().equals(info.activityInfo.name)) {
8         Class clz = Class.forName(info.activityInfo.name);
9         BroadcastReceiver bReceiver = (BroadcastReceiver) clz.newInstance();
10        bReceiver.onReceive(ctx, intent); } } }

```

Figure 12: Improving precision in RIPPLe by working with intent analysis. Here, \rightarrow denotes data-flow and \dashrightarrow denotes the class names discovered after considering the constraints denoted by \rightarrow .

Class `ReferralReceiver` is a `BroadcastReceiver`, a basic component in Android. In method `sendOtherBroadcastReceiver()`, an intent with action `com.android.vending.INSTALL_REFERRER` is created (line 3) and used to find all the `BroadcastReceivers` that can handle this intent (line 4). In lines 5 – 9, appropriate ones are created reflectively (lines 7 – 8), on which `onReceive()` is called (line 9).

RIPPLe handles the incomplete information caused by `Context`, which contains some global information about an app. As `ctx = null`, `info.activityInfo.name = null`. RIPPLe first applies `[C3-FORNAME]` to `Class.forName(info.activityInfo.name)` in line 7 and then `[C12-NEW]` to `clz.newInstance()` in line 8 to infer the reflectively created objects from the cast `BroadcastReceiver`. As `BroadcastReceiver` is often extended, `bReceiver` is made to point to 17 different types of objects with 15 being false positives.

If RIPPLe works with an advanced intent analysis, these 15 false positives may be all avoided. By assuming optimistically that `ctx` points to the `Context` of the current app, and modeling the call `queryBroadcastReceivers()` to return the

information about all the components that can handle the intent `criterion`, `infoList` will just contain the information pertaining to the three `BroadcastReceivers`, `ReferralReceiver`, `GrowMobileInstallReceiver` and `Tracker`. By also filtering out `ReferralReceiver` path-sensitively in line 6, RIPPLe will now be able to make `bReceiver` to point to only two objects precisely, one object of type `GrowMobileInstallReceiver` and one object of type `Tracker`, in line 8. A similar pattern also appears in the app *Cooking Fever*.

6.3 RQ3: Scalability

Table 2 compares the analysis times of RIPPLe, STRINF, and SPARK (SOOT’s pointer analysis without reflection analysis). For each app, the final harness that is iteratively constructed by FLOWDROID (to discover callbacks) is used as its main entry. For all the 17 apps except two (`canteenhd` and `angrybirds`), RIPPLe finishes in under 10 secs. For all the 17 apps combined, SPARK, STRINF and RIPPLe spend a total of 82.5, 86.0 and 93.4 seconds, respectively.

6.4 RQ4: Security Analysis

Table 2 also compares RIPPLe, STRINF and SPARK in terms of their effectiveness for enabling FLOWDROID to find sensitive data leaks in Android apps. For each analysis, FLOWDROID calls it iteratively to build a harness for an app (by modeling more and more callbacks discovered) until a fixed-point is reached. FLOWDROID will then perform a flow- and context-sensitive taint analysis on the inter-procedural CFG, which is constructed based on the call graph (CG) that is computed for the app with respect to the final harness obtained. Thus, FLOWDROID can be precise, in practice.

For each app, as shown in Table 2, the number of data leaks, together with sensitive source and sink calls, found by FLOWDROID under SPARK, STRINF and RIPPLe are compared. As discussed in Section 6.1, RIPPLe finds more true reflective targets than STRINF. For each app, RIPPLe’s CG is always a super-graph of STRINF’s CG, which is always a super-graph of SPARK’s CG. In particular, RIPPLe’s CG is larger than STRINF’s CG in 13 out of the 17 apps evaluated. As a result, FLOWDROID detects a total of 1,418, 1,486 and

1,796 leaks under SPARK, STRINF and RIPPLE, respectively.

Let us examine RIPPLE and STRINF in more detail. For 10 out of the 17 apps evaluated, FLOWDROID reports the same number of leaks for each app under both analyses. However, for the remaining seven apps, FLOWDROID reports 310 more leaks under RIPPLE than STRINF (highlighted by the numbers in bold in the last column of Table 2). RIPPLE’s ability in finding more true reflective targets in these apps than STRINF, as shown in Table 1, has certainly paid off.

Let us revisit two examples discussed in Section 3 to understand reflection-induced privacy violations. Let us consider the code taken from *Angry Birds* in Figure 2. Due to an undermined intent, `cName = null`. RIPPLE infers a total of five reflectively created objects for `Class.newInstance()` in line 6 based on its post-dominant cast `MMBaseActivity`. As discussed in Section 5.6, all these five objects are true targets configured to provide different forms of advertisement, enabling FLOWDROID to detect 49 leaks that are missed by STRINF, on the methods called directly or indirectly on these objects. For this entire app, FLOWDROID finds 54 more leaks under RIPPLE than STRINF. A similar code pattern also appears in *Dumb Ways to Die* where FLOWDROID finds 77 more leaks under RIPPLE than STRINF.

Let us now consider the code snippet taken from *Twist* in Figure 3. RIPPLE infers that `i()`, `d()`, `w()`, `e()`, `v()` and `wtf()` in class `Log`, where the last two are false positives, are the potential targets invoked at `logMtd.invoke(null, tag, msg)` in line 6. Some sensitive data may be accidentally passed to `msg` and get written to log files, resulting in potential security vulnerabilities. Due to the six target methods discovered, FLOWDROID finds a total of 12 data leaks (= 2 sources \times 6 sinks) from two sensitive sources, of which 4 from `v()` and `wtf()` are false positives.

7. RELATED WORK

We review only the most relevant work on reflection analysis for Android apps and Java programs.

Android Apps. Ernst et al. [10] presented CHECKER, a data-flow analysis for Android apps with reflective calls handled later [6]. As for the reflection resolution approach used, CHECKER performs regular string inference as DROIDRA for constant class and method names but requires user annotations to handle non-constant class and method names. In contrast, RIPPLE aims to automatically infer reflection targets at reflective calls according to the type information available.

Li et al. [17] introduced DROIDRA, a string inference analysis for resolving reflection in Android apps. In this work, the reflection resolution problem is reduced to one of solving a constant string propagation in the program. In DROIDRA, reflective calls can only be resolved if their class and method names are constants and ignored otherwise.

Rasthofer et al. [31] developed HARVESTER, an approach for automatically extracting runtime values from Android apps. HARVESTER takes an Android installation package and performs a backward slicing starting at a point of interest. Afterwards, a new, reduced package is generated and executed on a stock Android emulator or real phone to log the values of interest at runtime, such as some class and method names that are dynamically loaded and invoked via reflection. HARVESTER is designed to fight against code obfuscation techniques in order to extract runtime values, such

as class and method names, that may be first encrypted and then used in reflection calls under some particular inputs. In contrast, RIPPLE attempts to infer reflective targets used under these and other circumstances statically under all possible inputs. It will be interesting to investigate how to combine dynamic techniques such as HARVESTER and static reflection analyses such as RIPPLE to make a desired tradeoff among soundness, precision and scalability.

Zhauniarovich et al. [47] introduced STADYNA, a system that interleaves static and dynamic analysis in order to reveal the program behaviors caused by dynamic code update techniques, such as dynamic class loading and reflection. STADYNA requires a modified Android virtual machine to log the side-effects of program behaviors at runtime, including the targets accessed at reflective calls. This online analysis approach involves human efforts (e.g., in preparing for test inputs), but with no guarantee for code coverage. In contrast, RIPPLE is a static reflection analysis, enabling it to be integrated with a range of static security analyses, such as FLOWDROID, to achieve improved code coverage and soundness, as demonstrated in this paper.

Java Programs. There are several reflection analysis techniques for Java programs [18, 19, 21, 37]. Earlier, Livshits et al. [21] suggested to discover reflective targets by tracking the flow of string constants representing class/method/field names and infer reflective targets based on post-dominating type casts for `Class.newInstance()` calls if their class names are statically unknown strings. Recently, Li. et al. introduced ELF [18] and SOLAR [19] to apply sophisticated type inference to resolve reflective targets effectively. In particular, SOLAR is able to accurately identify where reflection is resolved unsoundly or imprecisely. In addition, it provides a mechanism to balance soundness, precision and scalability, representing a state-of-the-art solution for Java. A recent program slicing technique, called program tailoring [20], can also be leveraged to resolve reflection calls precisely. However, all the reflection analysis techniques proposed for Java cannot resolve a reflective call fully if the data-flows needed (e.g., class or method names) at the call are null.

Reflection analysis usually works together with pointer analysis in order to discover the targets at reflective calls. For pointer analysis techniques developed for Java programs, we refer to [16, 23–25, 34, 35, 38–40, 44].

8. CONCLUSION

Due to the ubiquity of mobile phones and the rapid development of other connected mobile devices (e.g., tablets and e-books), security vulnerabilities in Android apps, especially the presence of reflection, pose major security threats. In this paper, we introduce a reflection analysis for Android apps for discovering the behaviors of reflective calls, which can cause directly or indirectly security vulnerabilities such as privacy violations. We advance the state-of-the-art reflection analysis for Android apps, by (1) bringing forward the ubiquity of IIEs for static analysis, (2) introducing RIPPLE, the first IIE-aware reflection analysis, and (3) demonstrating that RIPPLE can resolve reflection in real-world Android apps precisely and efficiently, and consequently, improve the effectiveness of downstream Android security analyses.

In future work, we plan to combine RIPPLE with some dynamic analysis and integrate RIPPLE with an advanced pointer analysis to improve both soundness and precision.

Acknowledgement

This research is supported by an Australian Research Council grant, DP170103956.

9. REFERENCES

- [1] Virusshare project. <http://virusshare.com/>.
- [2] L. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, 1994.
- [3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and explainable detection of Android malware in your pocket. In *NDSS '14*.
- [4] S. Arzt and E. Bodden. Stubdroid: automatic inference of precise data-flow summaries for the Android framework. In *ICSE '16*, pages 725–735.
- [5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *PLDI '14*, pages 259–269.
- [6] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. D. Ernst, et al. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *ASE '15*, pages 669–679.
- [7] O. Bastani, S. Anand, and A. Aiken. Specification inference using context-free language reachability. In *POPL '15*, pages 553–566.
- [8] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *ICSE '14*, pages 175–186.
- [9] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder, and X. Jiang. Profiling user-trigger dependence for Android malware detection. *Computers & Security*, 49:255–273, 2015.
- [10] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhaskar, S. Han, P. Vines, and E. X. Wu. Collaborative verification of information flow for a high-assurance app store. In *CCS '14*, pages 1092–1104.
- [11] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *FSE '14*, pages 576–587.
- [12] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in Android applications. In *S&P '16*, pages 377–396.
- [13] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of Android applications in DroidSafe. In *NDSS '15*.
- [14] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS '12*.
- [15] S. Lee, J. Dolby, and S. Ryu. Hybridroid: static analysis framework for Android hybrid applications. In *ASE '16*, pages 250–261.
- [16] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? *CC*, pages 47–64, 2006.
- [17] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein. DroidRA: taming reflection to support whole-program analysis of Android apps. In *ISSTA '16*, pages 318–329.
- [18] Y. Li, T. Tan, Y. Sui, and J. Xue. Self-inferencing reflection resolution for Java. In *ECOOP '14*, pages 27–53.
- [19] Y. Li, T. Tan, and J. Xue. Effective soundness-guided reflection analysis. In *SAS '15*, pages 162–180.
- [20] Y. Li, T. Tan, Y. Zhang, and J. Xue. Program tailoring: slicing by sequential criteria. In *ECOOP '16*, pages 15:1–15:27.
- [21] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *APLAS '05*, pages 139–160.
- [22] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting Android apps for component hijacking vulnerabilities. In *CCS '12*, pages 229–240.
- [23] Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with CFL-reachability. *CC '13*, pages 61–81.
- [24] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. *ISSTA '12*, pages 1–11.
- [25] P. H. Nguyen and J. Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *ACSC '05*, pages 9–18.
- [26] J. Oberheide and C. Miller. Dissecting the Android bouncer. *SummerCon '12*.
- [27] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, and Y. Le Traon. Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis. In *POPL '16*, pages 469–484.
- [28] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel. Composite constant propagation: Application to Android inter-component communication analysis. In *ICSE '15*, pages 77–88.
- [29] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in Android: An essential step towards holistic security analysis. In *USENIX Security '13*, pages 543–558.
- [30] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden. Droidforce: enforcing complex, data-centric, system-wide policies in Android. In *ARES '14*, pages 40–49.
- [31] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime values in Android applications that feature anti-analysis techniques. In *NDSS '16*.
- [32] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *CODASPY '13*.
- [33] F. Ruiz. “fakeisntaller” leads the attack on Android phones. *McAfee Labs Website*, Oct, 2012.
- [34] L. Shang, Y. Lu, and J. Xue. Fast and precise points-to analysis with incremental CFL-reachability summarisation: Preliminary experience. In *ASE '12*, pages 270–273.
- [35] L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. In *CGO '12*, pages 264–274.

- [36] R. Slavin, X. Wang, M. B. Hosseini, J. Hester, R. Krishnan, J. Bhatia, T. D. Breau, and J. Niu. Toward a framework for detecting privacy policy violations in Android application code. In *ICSE '16*, pages 25–36.
- [37] Y. Smaragdakis, G. Balatsouras, G. Kastrinis, and M. Bravenboer. More sound static handling of Java reflection. In *APLAS '15*, pages 485–503.
- [38] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. *POPL*, pages 17–30, 2011.
- [39] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. *PLDI*, pages 387–400, 2006.
- [40] T. Tan, Y. Li, and J. Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. *SAS*, pages 489–510, 2016.
- [41] E. Tinaztepe, D. Kurt, and A. Güleç. Android obad. *Technical Analysis Paper*, 2013.
- [42] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: a Java bytecode optimization framework. In *CASCON '99*, page 13.
- [43] F. Wei, S. Roy, X. Ou, et al. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps. In *CCS '14*, pages 1329–1341.
- [44] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *PLDI*, pages 131–144, 2004.
- [45] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *ICSE '15*, pages 89–99.
- [46] M. Zhang, Y. Duan, Q. Feng, and H. Yin. Towards automatic generation of security-centric descriptions for Android apps. In *CCS '15*, pages 518–529.
- [47] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci. Stadya: addressing the problem of dynamic code updates in the security analysis of Android applications. In *CODASPY '15*, pages 37–48. ACM.
- [48] M. Zheng, M. Sun, and J. C. S. Lui. Droid analytics: A signature based analytic system to collect, extract, analyze and associate Android malware. In *TRUSTCOM '13*, pages 163–171.
- [49] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In *CODASPY '12*, pages 317–326.